What is claimed is:

1. A system for shortening the compiling time of byte codes in a Java program, comprising:

a class loader unit for loading byte codes generated by compiling Java program source codes;

a first memory unit for maintaining the byte codes loaded by the class loader unit and native codes generated by compiling the byte codes in an accessible state;

a second memory unit for storing the native codes that are loaded into the first memory unit in the accessible state;

a native code manager unit for searching the native codes stored in the second memory unit and loading requested native codes into the first memory unit according to a request by a class loader unit; and

an execution unit for executing the native codes that are loaded into the first memory unit in the accessible state.

2. The system as claimed in claim 1, further comprising a garbage collector unit for automatically collecting space occupied by unused codes in the first memory unit.

3. The system as claimed in claim 2, wherein the garbage collector unit requests the native code manager unit to store the native codes, which have been loaded into the first memory unit, in the second memory unit if a space shortage occurs in the first memory unit.

4. The system as claimed in claim 1, wherein the native code manager unit stores the native codes, which have been loaded into the first memory unit, in the second memory unit.

5. The system as claimed in claim 1, wherein the native code manager unit employs an LRU (least recently used) method to manage the native codes stored in the second memory unit.

6. The system as claimed in claim 4, wherein the native code manager unit employs an LRU (least recently used) method to manage the native codes stored in the second memory unit.

7. The system as claimed in the claim 1, wherein the execution unit comprises:

a byte code interpreter for interpreting the byte codes, which are loaded into the first memory unit in the accessible state, to be executed;

a runtime profiler for checking whether the byte codes being interpreted by the byte code interpreter are frequently used byte codes; and

a native code compiler for compiling the checked byte codes to native codes if the checked byte codes are determined as the frequently used byte codes by the runtime profiler.

8. A method for shortening the compiling time of byte codes in a Java program, comprising the steps of:

(a1) loading compiled byte codes by a class loader unit;

(a2) requesting a native code manager unit to search native codes corresponding to the loaded byte codes;

(a3) searching for the requested native codes in a second memory unit;

(a4) transmitting the requested native codes to a first memory unit; and

(a5) executing the transmitted native codes by a code execution unit.

9. The method as claimed in claim 8, wherein the native codes stored in the second memory unit are managed by the native code manager unit according to an LRU (least recently used) method.

10. The method as claimed in claim 8, further comprising the steps of, if it is determined from the search results that there are no corresponding native codes in the second memory unit:

(a6) transmitting the byte codes loaded by the class loader unit to the first memory unit; and

(a7) interpreting and executing the byte codes transmitted to the first memory unit by a byte code interpreter.

11. The method as claimed in claim 10, wherein step (a7) comprises the step of checking, by a runtime profiler, whether the byte codes being interpreted by the byte code interpreter are frequently used byte codes.

12. The method as claimed in claim 11, further comprising the steps of, if the byte codes are identified as frequently used byte codes from the check results:

(a8) generating, by a native code compiler, native codes corresponding to the frequently used byte codes by compiling the byte codes interpreted by the byte code interpreter;

(a9) loading the generated native codes into the first memory unit; and

(a10) storing the loaded native codes in the second memory unit by the native code manager unit.

13. The method as claimed in claim 8, wherein the native codes loaded into the first memory unit are stored in the second memory unit if the execution of the Java program is terminated or a space shortage occurs in the first memory unit.

14. The method as claimed in claim 10, wherein the native codes stored in the second memory unit are managed by the native code manager unit according to an LRU (least recently used) method.

15. The method as claimed in claim 13, wherein the native codes stored in the second memory unit are managed by the native code manager unit according to an LRU (least recently used) method.